A Framework for QoS-aware Execution of Workflows over the Cloud

Moreno Marzolla
Università di Bologna
Dipartimento di Scienze dell'Informazione
Mura A. Zamboni 7, I-40127 Bologna, Italy
Email: marzolla@cs.unibo.it

Raffaela Mirandola
Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, I-20133 Milano, Italy
Email: mirandola@elet.polimi.it

***Abstract*—The Cloud Computing paradigm is providing system architects with a new powerful tool for building scalable applications. Clouds allow allocation of resources on a "pay-as-you-go" model, so that additional resources can be requested during peak loads and released after that. However, this flexibility asks for appropriate dynamic reconfiguration strategies. In this paper we describe SAVER (qoS-Aware workflows oVER the Cloud), a QoS-aware algorithm for executing workflows involving Web Services hosted in a Cloud environment. SAVER allows execution of arbitrary workflows subject to response time constraints. SAVER uses a passive monitor to identify workload fluctuations based on the observed system response time. The information collected by the monitor is used by a planner component to identify the minimum number of instances of each Web Service which should be allocated in order to satisfy the response time constraint. SAVER uses a simple Queueing Network (QN) model to identify the optimal resource allocation. Specifically, the QN model is used to identify bottlenecks, and predict the system performance as Cloud resources are allocated or released. The parameters used to evaluate the model are those collected by the monitor, which means that SAVER does not require any particular knowledge of the Web Services and workflows being executed. Our approach has been validated through numerical simulations, whose results are reported in this paper.**

I. Introduction

The emerging Cloud computing paradigm is rapidly gaining consensus as an alternative to traditional IT systems, as exemplified by the Amazon EC2 [1], Xen [2], IBM Cloud [3], and Microsoft Cloud [4]. Informally, the Cloud computing paradigm allows computing resources to be seen as a utility, available on demand. The term "resource" may represent infrastructure, platforms, software, services, or storage. In this vision, the Cloud provider is responsible to make the resources available to the users as they request it.

Cloud services can be grouped into three categories [5]: Infrastructure as a Service (IaaS), providing low-level resources such as Virtual Machines (VMs) (e.g., Amazon EC2 [1]); Platform as a Service (PaaS), providing software development frameworks (e.g., Microsoft Azure [4]); and Software as a Service (SaaS), providing applications (e.g., Salesforce.com [6]).

The Cloud provider has the responsibility to manage the resources it provides (being them VM instances, programming frameworks or applications) so that the user requirements and the desired Quality of Service (QoS) are satisfied. Cloud users are usually charged according to the amount of resources they consume (e.g., some amount of money per hour of CPU usage). In this way, customers can avoid capital expenditures by using Cloud resources on a "pay-as-you-go" model.

Users QoS requirements (e.g., timeliness, availability, security) are usually the result of a negotiation process engaged between the resource provider and the user, which culminates in the definition of a Service Level Agreement (SLA) concerning their respective obligations and expectations. Guaranteeing SLAs under variable workloads for different application and service models is extremely challenging: Clouds are characterized by high load variance, and users have heterogeneous and competing QoS requirements.

In this paper we present SAVER (qoS-Aware workflows oVER the Cloud), a workflow engine provided as a SaaS. The engine allows different types of workflows to be executed over a set of Web Services (WSs). Workflows are described using some appropriate notations (e.g., using the WS-BPEL [7] workflow description language). The workflow engine takes care of interacting with the appropriate WSs as described in the workflow.

In our scenario, users can negotiate QoS requirements with the service provider; specifically, for each type c of workflow, the user may request that the average execution time of the whole workflow should not exceed a threshold R_c^+. Once the QoS requirements have been negotiated, the user can submit any number of workflows of the different types. Both the submission rate and the time spent by the workflows on each WS can fluctuate over time.

Traditionally, when deciding the amount of resources to be dedicated to applications, service providers considered worst-case scenarios, resulting in resource over-provisioning. Since the worst-case scenario rarely happens, a static system deployment results in a processing infrastructure which is largely under-utilized.

To increase the utilization of resources while meeting the requested SLA, SAVER uses an underlying IaaS Cloud to provide computational power on demand. The Cloud hosts multiple instances of each WS, so that the workload can be balanced across the instances. If a WS is heavily used, SAVER will increase the number of instances by requesting new resources from the Cloud. In this way, the response time

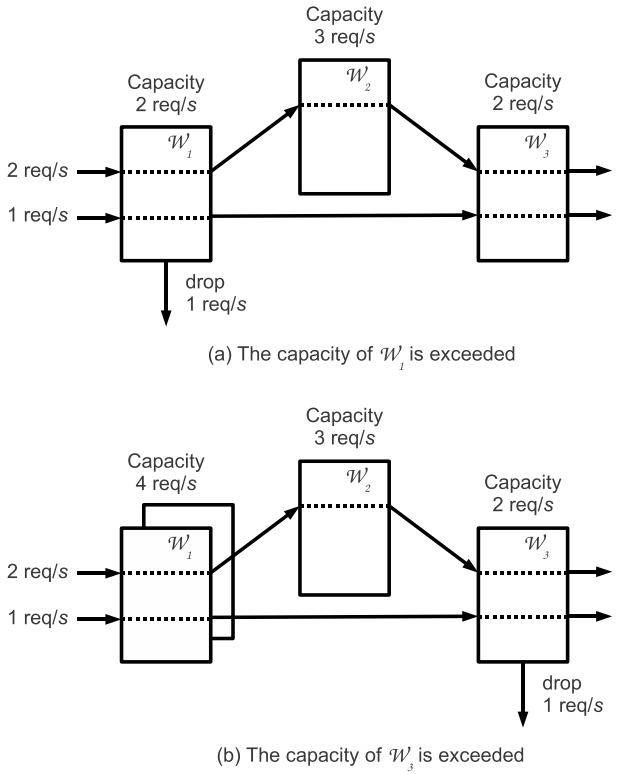

Fig. 1. Illustration of the bottleneck shift issue

of that WS can be reduced, reducing the total execution time of workflows as well. SAVER monitors the workflow engine and detects when some constraints are being violated. System reconfigurations are triggered periodically, when instances are added or removed where necessary.

Despite its conceptual simplicity, the idea above is quite challenging to implement in practice. To better illustrate the problem, let us consider the situation shown in Fig, 1, which is modeled upon a similar example from [8]. We have three Web Services $\mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3$ which are used by two types of workflows. Instances of the first type arrive at a rate of 2 req/s, and execute operations on $\mathcal{W}_1$, $\mathcal{W}_2$ and $\mathcal{W}_3$. Instances of the second workflow type arrive at a rate of 1 req/s and only use $\mathcal{W}_1$ and $\mathcal{W}_3$. Each WS has a maximum capacity, which corresponds to the maximum request rate it can handle. Web Services 1 and 3 have a maximum capacity of 2 req/s, while WS 2 has a capacity of 3 req/s.

In Fig. 1(a) the capacity of $\mathcal{W}_1$ is exceeded, because the aggregate arrival rate (3 req/s) is greater than its processing capacity. Thus, a queue of unprocessed invocations of $\mathcal{W}_1$ builds up, until requests start to timeout and are dropped at a rate of 1 req/s. To eliminate the bottleneck, a possible solution is to create multiple instances of the bottleneck WS on different servers, and balance the load across all instances. If we apply this strategy and create two instances of $\mathcal{W}_1$, we get the situation shown in Fig. 1(b): the aggregate processing capacity of $\mathcal{W}_1$ is now 4 req/s, and thus Web Service 1 is no longer the bottleneck. However, the bottleneck shifts to $\mathcal{W}_3$, which now sees an aggregate arrival rate of 3 req/s and has a capacity of 2 req/s.

The situation above demonstrates the *bottleneck shift* phenomenon: fixing a bottleneck may create another bottleneck at a different place. Thus, satisfying QoS constraints on systems subject to variable workloads is challenging, because identifying the system configuration which satisfies all constraints might involve multiple reconfigurations of individual components (in our scenario, adding WS instances). If the reconfiguration is implemented in a purely reactive manner, each step must be applied sequentially in order to monitor its impact and plan for the next step. This is clearly inefficient because adaptation would be exceedingly slow.

In general, the response time at a specific WS depends both on the number of instances of that Web Service, and also on the intensity of other workload classes (workflow types). Thus, a suitable system performance model must be used in order to predict the response time of a given configuration. The performance model can be used to drive the reconfiguration process proactively: different system configurations can be evaluated quickly, and multiple reconfiguration steps can be planned in advance. SAVER uses a open, multiclass Queueing Network (QN) model to represent resource contention by multiple independent request flows, which is crucial in our scenario. The parameters which are needed to evaluate the QN model can be easily obtained by passively monitoring the running system. The performance model is used within a greedy strategy which identifies an approximate solution to the optimization problem minimizing the number of WS instances while respecting the SLA.

Structure of this paper: The remainder of this paper is organized as follows. In Section II we review the scientific literature and compare SAVER with related works. In Section III we give a precise formulation of the problem we are addressing. In Section IV we describe the Queueing Network performance model of the Cloud-based workflow engine. SAVER will be fully described in Section V, including the high-level architecture and the details of the reconfiguration algorithms. The effectiveness of SAVER have been evaluated by means of simulation experiments, whose results will be discussed in Section VI. Finally, conclusions and future works are presented in Section VII. In order to make this paper self-contained without sacrificing clarify, we relegated the mathematical details of the analysis of the performance model in a separate Appendix.

II. Related works

Several research contributions have previously addressed the issue of optimizing the resource allocation in cluster-based service centers. Recently, with the emerging of virtualization approaches and Cloud computing, additional research on automatic resource management has been conducted. In this section we briefly review some recent results; some of them take advantage of control theory-based feedback loops [9],

[10], machine learning techniques [11], [12], or utility-based optimization techniques [13], [14].

When moving to virtualized environments the resource allocation problem becomes even more complex because of the introduction of virtual resources [14]. Several approaches have been proposed for QoS and resource management at run-time [9], [15]–[19].

The approach presented in [15] describes a method for achieving optimization in Clouds by using performance models all along the development and operation of the applications running in the Cloud. The proposed optimization aims at maximizing profits in the Cloud by guaranteeing the QoS agreed in the SLAs taking into account a large variety of workloads. A layered Cloud architecture taking into account different stakeholders is presented in [9]. The architecture supports self-management based on adaptive feedback control loops, present at each layer, and on a coordination activity between the different loops. Mistral [16] is a resource managing framework with a multi-level resource allocation algorithm considering reallocation actions based mainly on adding, removing and/or migrating virtual machines, and shutdown or restart of hosts. This approach is based on the usage of Layered Queuing Network (LQN) performance model. It tries to maximize the overall utility taking into account several aspects like power consumption, performance and transient costs in its reconfiguration process. In [18] the authors present an approach to self-adaptive resource allocation in virtualized environments based on online architecture-level performance models. The online performance prediction allow estimation of the effects of changes in user workloads and of possible reconfiguration actions. Yazir *et al.* [19] introduces a distributed approach for dynamic autonomous resource management in computing Clouds, performing resource configuration using through Multiple Criteria Decision Analysis.

With respect to these works, SAVER lies in the same research line fostering the usage of models at runtime to drive the QoS-based system adaptation. SAVER uses an efficient modeling and analysis technique that can then be used at runtime without undermining the system behavior and its overall performance.

Ferretti *et al.* propose in [17] a middleware architecture enabling a SLA-driven dynamic configuration, management and optimization of Cloud resources and services. The approach makes use of a load balancer that distributes the workload among the available resources. When the perceived QoS deviates from the SLA, the platform is dynamically reconfigured by acquiring new resources from the Cloud. On the other hand, if resources under-utilization is detected, the system triggers a reconfiguration to release those unused resources. This approach is purely reactive and considers a single-tier application, while SAVER works for an arbitrary number of WSs and uses a performance model to plan complex reconfigurations in a single step.

Canfora *et al.* [20] describe a QoS-aware service discovery and late-binding mechanism which is able to automatically adapt to changes of QoS attributes in order to meet the SLA. The authors consider the execution of workflows over a set of WSs, such that each WS has multiple functionally equivalent implementations. Genetic Algorithms are use to bind each WS to one of the available implementations, so that a fitness function is maximized. The binding is done at run-time, and depends on the values of QoS attributes which are monitored by the system. It should be observed that in SAVER we consider a different scenario, in which each WS has just one implementation which however can be instantiated multiple times. The goal of SAVER is to satisfy a specific QoS requirement (mean execution time of workflows below a given threshold) with the minimum number of instances.

III. Problem Formulation

SAVER is a workflow engine whose general structure is depicted in Fig. 2: it receives workflows from external clients, and executes them over a set of K WS $\mathcal{W}_1, \ldots, \mathcal{W}_K$. Workflows can be of C different types (or classes); for each class $c = 1, \ldots, C$, clients define a maximum allowed completion time R_c^+. This means that an instance of class c workflow must be completed, on average, in time less than R_c^+. New workflow classes can be created at any time; when a new class is created, its maximum response time is negotiated with the workflow service provider.

We denote with λ_c the average arrival rate of class c workflows. Arrival rates can change over time[1]. Since all WSs are shared between the workflows, the completion time of a workflow depends both on arrival rates $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_C)$, and on the utilization of each WS.

In order to satisfy the response time constraints, the system must adapt to cope with fluctuations of the workload. To do so, SAVER relies on a IaaS Cloud which maintains multiple instances of each WS. Run-time monitoring information is sent by all WSs back to the workflow engine to drive the adaptation process. We denote with N_k the number of instances of WS $\mathcal{W}_k$; a system configuration $\mathbf{N} = (N_1, \ldots, N_K)$ is an integer vector representing the number of allocated instances of each WS.

When a workflow interacts with $\mathcal{W}_k$, it is bound to one of the N_k instances so that the requests are evenly distributed. When the workload intensity increases, additional instances are created to eliminate the bottlenecks; when the workload decreases, surplus instances are shut down and released.

The goal of SAVER is to minimize the total number of WS instances while maintaining the mean execution time of type c workflows below the threshold R_c^+, $c = 1, \ldots, C$. Formally, we want to solve the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{N}) = \sum_{k=1}^{K} N_k \\ \text{subject to} \quad & R_c(\mathbf{N}) \leq R_c^+ \qquad \text{for all } c = 1, 2, \ldots, C \\ & N_i \in \{1, 2, 3, \ldots\} \end{aligned} \tag{1}$$

[1] In order to simplify the notation, we write λ_c instead of $\lambda_c(t)$. In general, we will omit explicit reference to t for all time-dependent parameters.

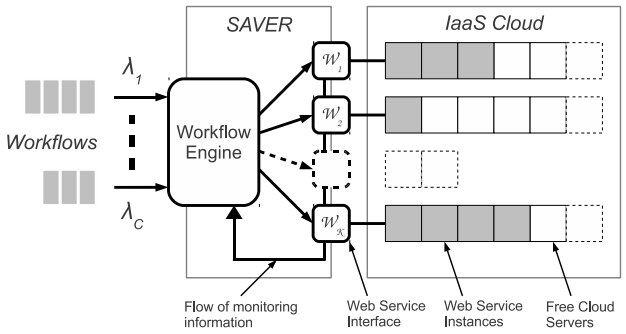

Fig. 2. System model

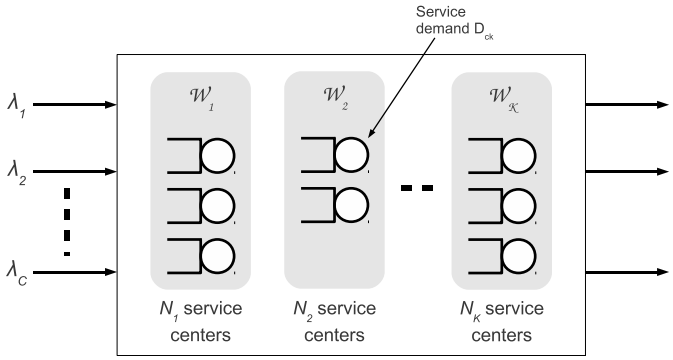

Fig. 3. Performance model based on an open, multiclass Queueing Network

where $R_c(\mathbf{N})$ is the mean execution time of type c workflows when the system configuration is $\mathbf{N} = (N_1, \ldots, N_K)$.

If the IaaS Cloud which hosts WS instances is managed by some third-party organization, then reducing the number of active instances reduces the cost of the workflow engine.

IV. System Performance Model

Before illustrating the details of SAVER, it is important to describe the QN performance model which is used to plan a system reconfiguration. We model the system of Fig. 2 using the open, multiclass QN model [21] shown in Fig. 3. A QN model is a set of queueing centers, which in our case are FIFO queues attached to a single server. Each server represents a single WS instance; thus, $\mathcal{W}_k$ is represented by N_k queueing centers, for each $k = 1, \ldots, K$. N_k can change over time, as resources are added or removed from the system.

In our QN model there are C different classes of requests, which are generated outside the system. Each request represents a workflow, thus workflow types are directly mapped to QN request classes. In order to simplify the analysis of the model, we make the simplifying assumption that the inter-arrival time of class c requests is exponentially distributed with arrival rate λ_c. This means that a new workflow of type c is submitted, on average, every $1/\lambda_c$ time units.

The interaction of a type c workflow with WS $\mathcal{W}_k$ is modeled as a visit of a class c request to one of the N_k queueing centers representing $\mathcal{W}_k$. We denote with $R_{ck}(\mathbf{N})$ the total time (*residence time*) spent by type c workflows on one of the N_k instances of $\mathcal{W}_k$ for a given configuration $\mathbf{N}$. The residence time is the sum of two terms: the *service demand* $D_{ck}(\mathbf{N})$ (average time spent by a WS instance executing the request) and queueing delay (time spent by a request in the waiting queue). The QN model allows multiple visits to the same queueing center, because the same WS can be executed multiple times by the same workflow. The residence time and service demands are the sum of residence and service time of *all* invocations of the same WS instance.

The *utilization* $U_k(\mathbf{N})$ of an instance of $\mathcal{W}_k$ is the fraction of time the instance is busy processing requests. If the workload is evenly balanced, then both the residence time $R_{ck}(\mathbf{N})$ and the utilization $U_k(\mathbf{N})$ are almost the same for all N_k instances of $\mathcal{W}_k$.

TABLE I
Symbols used in this paper

C	Number of workflow types
K	Number of Web Services
$\boldsymbol{\lambda}$	Vector of per-class Arrival rates
$\mathbf{M}$	Current system configuration
$\mathbf{N}, \mathbf{N}'$	Arbitrary system configurations
$R_{ck}(\mathbf{N})$	Residence time of type c workflows on an instance of $\mathcal{W}_k$
$D_{ck}(\mathbf{N})$	Service demand of type c workflows on an instance of $\mathcal{W}_k$
$R_c(\mathbf{N})$	Response time of type c workflows
$U_k(\mathbf{N})$	Utilization of an instance of $\mathcal{W}_k$
R_c^+	Maximum allowed response time for type c workflows

Table I summarizes the symbols used in this paper.

V. Architectural Overview of SAVER

SAVER is a reactive system based on the Monitor-Analyze-Plan-Execute (MAPE) control loop shown in Fig. 4. During the *Monitor* step, SAVER collects operational parameters by observing the running system. The parameters are evaluate during the *Analyze* step; if the system needs to be reconfigured (e.g., because the observed response time of class c workflows exceeds the threshold R_c^+, for some c), a new configuration is identified in the *Plan* step. We use the QN model described in Section IV to evaluate different configurations and identify an optimal server allocation such that all QoS constraints are satisfied. Finally, during the *Execute* step, the new configuration is applied to the system: WS instances are created or destroyed as needed by leveraging the IaaS Cloud. Unlike other reactive systems, SAVER can plan complex reconfigurations, involving multiple additions/removals of resources, in a single step.

A. Monitoring System Parameters

The QN model is used to estimate the execution time of workflow types for different system configurations. To analyze the QN it is necessary to know two parameters: (*i*) the arrival rate of type c workflows, λ_c, and (*ii*) the service demand $D_{ck}(\mathbf{M})$ of type c workflows on an instance of WS $\mathcal{W}_k$, for the current configuration $\mathbf{M}$.

The parameters above can be computed by monitoring the system over a suitable period of time. The arrival rates λ_c can be estimated by counting the number A_c or arrivals of type c workflows which are submitted over the observation period of length T. Then λ_c can be defined as $\lambda_c = A_c/T$.

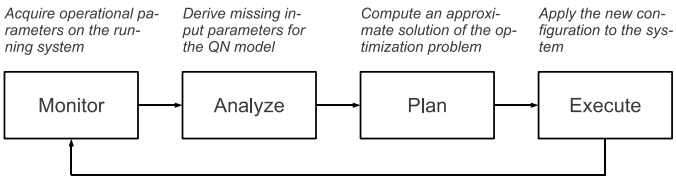

Fig. 4. SAVER Control Loop

TABLE II
EQUATIONS FOR THE QN MODEL OF FIG. 3

$$U_k(\mathbf{N}) = \sum_{c=1}^{C} \lambda_c D_{ck}(\mathbf{N}) \quad (2)$$

$$R_{ck}(\mathbf{N}) = \frac{D_{ck}(\mathbf{N})}{1 - U_k(\mathbf{N})} \quad (3)$$

$$R_c(\mathbf{N}) = \sum_{k=1}^{K} N_k R_{ck}(\mathbf{N}) \quad (4)$$

Measuring the service demands $D_{ck}(\mathbf{M})$ is a bit more difficult because they must not include the time spent by a request waiting to start service. If the WSs do not provide detailed timing information (e.g., via their execution logs), it is possible to estimate $D_{ck}(\mathbf{M})$ from parameters which can be easily observed by the workflow engine, that are the measured residence time $R_{ck}(\mathbf{M})$ and utilization $U_k(\mathbf{M})$. We use the equations shown in Table II, which hold for the open multiclass QN model in Fig. 3. These equations describe well known properties of open QN models, so they are given here without any proof. The interested reader is referred to [21] for details.

The residence time is the total time spent by a type c workflow with one instance of WS $\mathcal{W}_k$, including waiting time and service time. The workflow engine can measure $R_{ck}(\mathbf{M})$ as the time elapsed from the instant a type c workflow sends a request to one of the N_k instances of $\mathcal{W}_k$, to the time the request is completed. The utilization $U_k(\mathbf{M})$ of an instance of $\mathcal{W}_k$ can be obtained by the Cloud service dashboard (or measured on the computing nodes themselves). Using (3) the service demands can be expressed as

$$D_{ck}(\mathbf{M}) = R_{ck}(\mathbf{M})\,(1 - U_k(\mathbf{M})) \quad (5)$$

B. Finding a new configuration

In order to find an approximate solution to the optimization problem (1), SAVER starts from the current configuration $\mathbf{M}$, which may violate some response time constraints, and executes Algorithm 1. After collecting device utilizations, response times and arrival rates, SAVER estimates the service demands D_{ck} using Eq. (5).

Then, SAVER identifies a new configuration $\mathbf{N} \geq \mathbf{M}$[2] by calling the function ACQUIRE(). The new configuration $\mathbf{N}$ is computed by greedily adding new instances to bottleneck WSs.

[2] $\mathbf{N} \geq \mathbf{M}$ iff $N_k \geq M_k$ for all $k = 1, \ldots, K$

Algorithm 1 The SAVER Algorithm

Require: R_c^+: Maximum response time of type c workflows
1: Let $\mathbf{M}$ be the initial configuration
2: **loop**
3: Monitor $R_{ck}(\mathbf{M})$, $U_k(\mathbf{M})$, λ_c
4: **for all** $c := 1, \ldots, C;\ k := 1, \ldots, K$ **do**
5: Compute $D_{ck}(\mathbf{M})$ using Eq. (5)
6: $\mathbf{N} := \mathsf{Acquire}(\mathbf{M}, \boldsymbol{\lambda}, \mathbf{D}(\mathbf{M}), \mathbf{U}(\mathbf{M}))$
7: **for all** $c := 1, \ldots, C;\ k := 1, \ldots, K$ **do**
8: Compute $D_{ck}(\mathbf{N})$ and $U_k(\mathbf{N})$ using Eq. (7) and (8)
9: $\mathbf{N}' := \mathsf{Release}(\mathbf{N}, \boldsymbol{\lambda}, \mathbf{D}(\mathbf{N}), \mathbf{U}(\mathbf{N}))$
10: Apply the new configuration $\mathbf{N}'$ to the system
11: $\mathbf{M} := \mathbf{N}'$ {Set $\mathbf{N}'$ as the current configuration $\mathbf{M}$}

The QN model is used to estimate response times as instances are added: no actual resources are instantiated from the Cloud service at this time.

The configuration $\mathbf{N}$ returned by the function ACQUIRE() does not violate any constraint, but might contain too many WS instances. Thus, SAVER invokes the function RELEASE() which computes another configuration $\mathbf{N}' \leq \mathbf{N}$ by removing redundant instances, ensuring that no constraint is violated. To call procedure RELEASE() we need to estimate the service demands $D_{ck}(\mathbf{N})$ and utilizations $U_k(\mathbf{N})$ with configuration $\mathbf{N}$. These can be easily computed from the measured values for the current configuration $\mathbf{M}$.

After both steps above, $\mathbf{N}'$ becomes the new current configuration: WS instances are created or terminated where necessary by acquiring or releasing hosts from the Cloud infrastructure.

Let us illustrate the functions ACQUIRE() and RELEASE() in detail.

a) Adding instances: Function ACQUIRE() is described by Algorithm 2. Given the system parameters and configuration $\mathbf{N}$, which might violate some or all response time constraints, the function returns a new configuration $\mathbf{N}'$ which is estimated not to violate any constraint. At each iteration, we identify the class b whose workflows have the maximum relative violation of the response time limit (line 2); response times are estimated using Eq. (9) in the Appendix. Then, we identify the WS $\mathcal{W}_j$ such that adding one more instance to it produces the maximum reduction in the class b response time (line 3). The configuration $\mathbf{N}$ is then updated by adding one instance to $\mathcal{W}_j$ (line 4); the updated configuration is $\mathbf{N} + \mathbf{1}_j$[3]. The loop terminates when no workload type is estimated to violate its response time constraint.

Termination of Algorithm 2 is guaranteed by the fact that function $R_c(\mathbf{N})$ is monotonically decreasing (Lemma 1 in the Appendix). Thus, $R_c(\mathbf{N} + \mathbf{1}_j) < R_c(\mathbf{N})$ for all c.

b) Removing instances: The function RELEASE(), described by Algorithm 3, is used to deallocate (release) WS instances from an initial configuration $\mathbf{N}$ which does not

[3] $\mathbf{1}_j$ is a vector with K elements, whose j-th element is one and all others are set to zero

Algorithm 2 Acquire$(\mathbf{N}, \boldsymbol{\lambda}, \mathbf{D}(\mathbf{N}), \mathbf{U}(\mathbf{N})) \rightarrow \mathbf{N}'$

Require: $\mathbf{N}$ System configuration
Require: $\boldsymbol{\lambda}$ Current arrival rates of workflows
Require: $\mathbf{D}(\mathbf{N})$ Service demands at configuration $\mathbf{N}$
Require: $\mathbf{U}(\mathbf{N})$ Utilizations at configuration $\mathbf{N}$
Ensure: $\mathbf{N}$ New system configuration
1: **while** $\left(R_c(\mathbf{N}) > R_c^+ \text{ for any } c\right)$ **do**
2: $\quad b := \arg\max_c \left\{ \dfrac{R_c(\mathbf{N}) - R_c^+}{R_c^+} \;\middle|\; c = 1, \ldots, C \right\}$
3: $\quad j := \arg\max_k \{ R_b(\mathbf{N}) - R_b(\mathbf{N} + \mathbf{1}_k) \mid k = 1, \ldots, K \}$
4: $\quad \mathbf{N} := \mathbf{N} + \mathbf{1}_j$
5: **Return** $\mathbf{N}$

violate any response time constraint. The function implements a greedy strategy, in which a WS $\mathcal{W}_j$ is selected at each step, and its number of instances is reduced by one. Reducing the number of instances N_j of $\mathcal{W}_j$ is not possible if, either (*i*) the reduction would violate some constraint, or (*ii*) the reduction would cause the utilization of some WS instances to become greater than one (see Eq. (11) in the Appendix).

We start by defining the set S containing the index of WSs whose number of instances can be reduced without exceeding the processing capacity (line 3). Then, we identify the workflow class d with the maximum (relative) response time (line 5). Finally, we identify the value $j \in S$ such that removing one instance of $\mathbf{W}_j$ produces the minimum increase in the response time of class d workflows (line 6). The rationale is the following. Type d workflows are the most likely to be affected by the removal of one WS instance, because their relative response time (before the removal) is the highest among all workflow types. Once the "critical" class d has been identified, we try to remove an instance from the WS j which causes the smallest increase of class d response time. Since response time increments are additive (see Appendix), if the removal of an instance of $\mathcal{W}_j$ violates some constraints, no further attempt should be done to consider $\mathcal{W}_j$, and we remove j from the candidate set S.

From the discussion above, we observe that function RELEASE() computes a *Pareto-optimal* solution $\mathbf{N}$. This means that there exists no solution $\mathbf{N}' \leq \mathbf{N}$ such that $R_c(\mathbf{N}') \leq R_c^+$.

VI. Numerical Results

We performed a set of numerical simulation experiments to assess the effectiveness of SAVER; results will be described in this section. We implemented Algorithms 1, 2 and 3 using GNU Octave [22], an interpreted language for numerical computations.

In the first experiment we considered $K = 10$ Web Services and $C = 5$ workflow types. Service demands D_{ck} have been randomly generated, in such a way that class c workflows have service demands which are uniformly distributed in $[0, c/C]$. Thus, class 1 workflows have lowest average service demands, while type C workflows have highest demands. The system has been simulated for $T = 200$ discrete steps $t = 1, \ldots, T$;

Algorithm 3 Release$(\mathbf{N}, \boldsymbol{\lambda}, \mathbf{D}(\mathbf{N}), \mathbf{U}(\mathbf{N})) \rightarrow \mathbf{N}'$

Require: $\mathbf{N}$ System configuration
Require: $\boldsymbol{\lambda}$ Current arrival rates of workflows
Require: $\mathbf{D}(\mathbf{N})$ Service demands at configuration $\mathbf{N}$
Require: $\mathbf{U}(\mathbf{N})$ Utilizations at configuration $\mathbf{N}$
Ensure: $\mathbf{N}'$ New system configuration
1: **for all** $k := 1, \ldots, K$ **do**
2: $\quad Nmin_k := N_k \sum_{c=1}^{C} \lambda_c D_{ck}(\mathbf{N})$
3: $S := \{k \mid N_k > Nmin_k\}$
4: **while** $(S \neq \emptyset)$ **do**
5: $\quad d := \arg\min_c \left\{ \dfrac{R_c^+ - R_c(\mathbf{N})}{R_c^+} \;\middle|\; c = 1, \ldots, C \right\}$
6: $\quad j := \arg\min_k \left\{ R_c(\mathbf{N} - \mathbf{1}_k) - R_c^+ \;\middle|\; k \in S \right\}$
7: $\quad$ **if** $\left(R_c(\mathbf{N} - \mathbf{1}_j) > R_c^+ \text{ for any } c\right)$ **then**
8: $\quad\quad S := S \setminus \{j\}$ {No instance of $\mathcal{W}_j$ can be removed}
9: $\quad$ **else**
10: $\quad\quad \mathbf{N} := \mathbf{N} - \mathbf{1}_j$
11: $\quad\quad$ **if** $(N_j = Nmin_j)$ **then**
12: $\quad\quad\quad S := S \setminus \{j\}$
13: **Return** $\mathbf{N}$

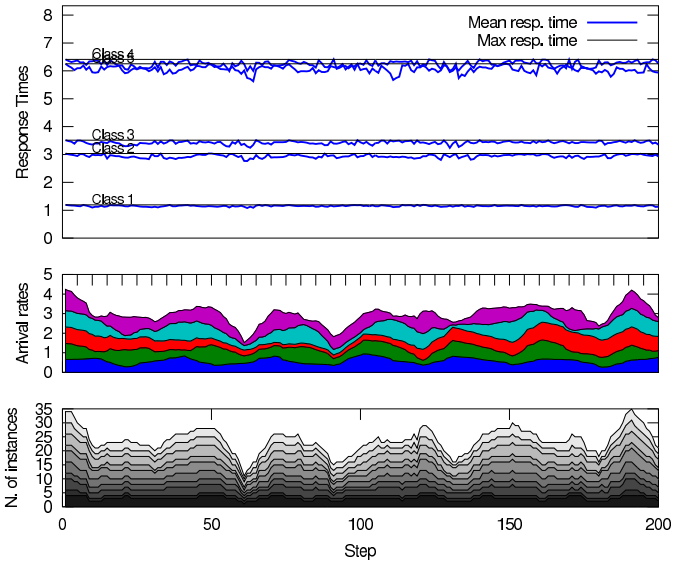

Fig. 5. Simulation results

each step corresponds to a time interval of length W, long enough to amortize the reconfiguration costs.

Arrival rates $\boldsymbol{\lambda}(t)$ at step t have been generated according to a fractal model, starting from a randomly perturbed sinusoidal pattern to mimic periodic fluctuations. Each workflow type has a different period.

Figure 5 shows the results of the simulation. The top part of the figure shows the estimated response time $R_c(\mathbf{N})$ (thick lines) and upper limit R_c^+ (thin horizontal lines) for each class $c = 1, \ldots, C$. The middle part of the figure shows the arrival rates $\lambda_c(t)$ for each class $c = 1, \ldots, C$; note that arrival rates have been stacked for clarity, such that the height of each individual band corresponds to the value $\lambda_c(t)$ from $c = 1$ (bottom) to $c = 5$ (top). The total height of the middle graph

is the total arrival rate of all workflow types. Finally, each band of the bottom part of Figure 5 shows the number N_k of instances of WS $\mathcal{W}_k$, from $k = 1$ (bottom) to $k = 10$ (top); again, the height of all areas represents the total number of resources which are allocated at each simulation step. As can be seen, the number of allocated resources closely follows the workload pattern.

We performed additional experiments in order to assess the efficiency of allocations produced by SAVER. In particular, we are interested in estimating the reduction in the number of allocated instances produced by SAVER. To do so, we considered different scenarios for all combinations of $C \in \{10, 15, 20\}$ workflow types and $K \in \{20, 40, 60\}$ Web Services. Each simulation has been executed for $T = 200$ steps; everything else (requests arrival rates, service demands) have been generated as described above.

Results are reported in Table III. Columns labeled C and K show the number of workflow types and Web Services, respectively. Columns labeled *Iter.* ACQUIRE*()* contain the maximum and average number of iterations performed by procedure ACQUIRE() (Algorithm 2); columns labeled *Iter.* RELEASE*()* contain the same information for procedure RELEASE() (Algorithm 3). Then, we report the minimum, maximum and total number of resources allocated by SAVER during the simulation run. Formally, let S_t denote the total number of WS instances allocated at simulation step t; then

$$\begin{aligned} \text{Min. instances} &= \min_t \{S_t\} \\ \text{Max. instances} &= \max_t \{S_t\} \\ \text{Total instances} &= \sum_t S_t \end{aligned}$$

Column labeled *WS Instances (static)* shows the number of instances that would have been allocated by provisioning for the worst case scenario; this value is simply $T \times \max_t\{S_t\}$. The last column shows the ratio between the total number of WS instances allocated by SAVER, and the number of instances that would have been allocated by a static algorithm to satisfy the worst-case scenario; lower values are better.

The results show that SAVER allocates between 64%–72% of the instances required by the worst-case scenario. As previously observed, if the IaaS provider charges a fixed price for each instance allocated at each simulation step, then SAVER allows a consistent reduction of the total cost, while still maintaining the negotiated SLA.

VII. CONCLUSIONS AND FUTURE WORKS

In this paper we presented SAVER, a QoS-aware algorithm for executing workflows involving Web Services hosted in a Cloud environment. The idea underlying SAVER is to selectively allocate and deallocate Cloud resources to guarantee that the response time of each class of workflows is less than a negotiated threshold. The capability of driving the dynamic resource allocation is achieved though the use of a feedback control loop. A passive monitor collects information that is used to identify the minimum number of instances of each WS which should be allocated to satisfy the response time constraints. The system performance at different configurations is estimated using a QN model; the estimates are used to feed a greedy optimization strategy which produces the new configuration which is finally applied to the system. Simulation experiments show that SAVER can effectively react to workload fluctuations by acquiring/releasing resources as needed.

The methodology proposed in this paper can be improved along several directions. In particular, in this paper we assumed that all requests of all classes are evenly distributed across the WS instances. While this assumption makes the system easier to analyze and implement, more effective allocations could be produced if we allow individual workflow classes to be routed to specific WS instances. This extension would add another level of complexity to SAVER, which at the moment is under investigation. We are also exploring the use of forecasting techniques as a mean to trigger resource allocation and deallocation proactively. Finally, we are working on the implementation of our methodology on a real testbed, to assess its effectiveness through a more comprehensive set of real experiments.

APPENDIX

Let $\mathbf{M}$ be the current system configuration; let us assume that, under configuration $\mathbf{M}$, the observed arrival rates are $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_C)$ and service demands are $D_{ck}(\mathbf{M})$. Then, for an arbitrary configuration $\mathbf{N}$, we can combine Equations (3) and (4) to get:

$$R_c(\mathbf{N}) = \sum_{k=1}^{K} N_k \frac{D_{ck}(\mathbf{N})}{1 - U_k(\mathbf{N})} \tag{6}$$

The current *total* class c service demand on all instances of $\mathcal{W}_k$ is $M_k D_{ck}(\mathbf{M})$, hence we can express service demands and utilizations of individual instances for an arbitrary configuration $\mathbf{N}$ as:

$$D_{ck}(\mathbf{N}) = \frac{M_k}{N_k} D_{ck}(\mathbf{M}) \tag{7}$$

$$U_k(\mathbf{N}) = \frac{M_k}{N_k} U_k(\mathbf{M}) \tag{8}$$

Thus, we can rewrite (6) as

$$R_c(\mathbf{N}) = \sum_{k=1}^{K} \frac{D_{ck}(\mathbf{M}) M_k N_k}{N_k - U_k(\mathbf{M}) M_k} \tag{9}$$

which allows us to estimate the response time $R_c(\mathbf{N})$ of class c workflows, given information collected by the monitor for the current configuration $\mathbf{M}$.

From (2) and (7) we get:

$$U_k(\mathbf{N}) = \frac{M_k}{N_k} \sum_{c=1}^{C} \lambda_c D_{ck}(\mathbf{M}) \tag{10}$$

TABLE III
SIMULATION RESULTS FOR DIFFERENT SCENARIOS

		Iter. ACQUIRE()		Iter. RELEASE()		WS Instances (dynamic)				
C	K	*max*	*avg*	*max*	*avg*	*min*	*max*	*tot*	WS Instances (static)	Dynamic/Static
10	20	14	1.30	15	2.53	36	127	16589	25400	0.65
10	40	22	2.43	19	3.81	76	257	33103	51400	0.64
10	60	35	3.54	35	5.12	122	378	50211	75600	0.66
15	20	10	1.27	13	2.56	78	178	23536	35600	0.66
15	40	23	2.20	26	3.68	138	340	44843	68000	0.66
15	60	34	3.20	44	5.04	239	526	68253	105200	0.65
20	20	9	1.19	13	2.50	114	206	28792	41200	0.70
20	40	24	2.33	29	4.00	215	408	57723	81600	0.71
20	60	21	3.00	30	4.89	347	602	86684	120400	0.72

Since by definition the utilization of any WS instance must be less than one, we can use (10) to define a lower bound on the number N_k of instances of $\mathcal{W}_k$ as:

$$N_k \geq M_k \sum_{c=1}^{C} \lambda_c D_{ck}(\mathbf{M}) \tag{11}$$

The following lemma can be easily proved:

Lemma 1: The response time function $R_c(\mathbf{N})$ is monotonically decreasing: for any two configurations $\mathbf{N}'$ and $\mathbf{N}''$ such that $N'_k \leq N''_k$ for all $k = 1, \ldots, K$, we have that $R_c(\mathbf{N}') \geq R_c(\mathbf{N}'')$

Proof: If we extend $R_c(\mathbf{N})$ to be a continuous function, its partial derivative is

$$\frac{\partial R_c}{\partial N_k} = \frac{-M_k^2 U_k(\mathbf{M}) D_{ck}(\mathbf{M})}{\left(N_k - U_k(\mathbf{M}) M_k\right)^2} \tag{12}$$

which is less than zero for any k for which the utilization $U_k(\mathbf{M})$ and service demand $D_{ck}(\mathbf{M})$ are nonzero. Hence, function $R_c(\mathbf{N})$ is decreasing. ∎

Note that, according to Eq. (9), response time increments are additive. This means that $R_c(\mathbf{N}) - R_c(\mathbf{N}+\mathbf{1}_j) = \mathbf{\Delta}_j$ and $R_c(\mathbf{N}) - R_c(\mathbf{N}+\mathbf{1}_i) = \mathbf{\Delta}_i$ imply $R_c(\mathbf{N}) - R_c(\mathbf{N}+\mathbf{1}_i+\mathbf{1}_j) = \mathbf{\Delta}_i + \mathbf{\Delta}_j$

REFERENCES

[1] "Amazon Elastic Compute Cloud (Amazon EC2)." [Online]. Available: http://aws.amazon.com/ec2/

[2] "Xen Cloud Platform." [Online]. Available: http://www.xen.org/

[3] "IBM Smart Cloud." [Online]. Available: http://www.ibm.com/ibm/cloud/

[4] "Windows Azure." [Online]. Available: http://www.microsoft.com/azure

[5] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 2010.

[6] "Salesforce CRM." [Online]. Available: http://www.salesforce.com/platform/

[7] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bolch, F. Curbera, M. Ford, Y. Goland, A. Guizar, N. Kartha, C. K. Lui, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, "Web services business process execution language version 2.0," OASIS Standard, Apr.7 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[8] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, 2008.

[9] M. Litoiu, M. Woodside, J. Wong, J. Ng, and G. Iszlai, "A business driven cloud optimization architecture," in *Proc. of the 2010 ACM Symp. on Applied Computing*, ser. SAC '10. ACM, 2010, pp. 380–385.

[10] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *ICAC*. ACM, 2009, pp. 117–126.

[11] J. O. Kephart, H. Chan, R. Das, D. W. Levine, G. Tesauro, F. L. R. III, and C. Lefurgy, "Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs," in *ICAC*. IEEE Computer Society, 2007, p. 24.

[12] R. Calinescu, "Resource-definition policies for autonomic computing," in *ICAS*. IEEE Computer Society, 2009, pp. 111–116.

[13] B. Urgaonkar, G. Pacifici, P. J. Shenoy, M. Spreitzer, and A. N. Tantawi, "Analytic modeling of multitier internet applications," *TWEB*, vol. 1, no. 1, 2007.

[14] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, "1000 islands: an integrated approach to resource management for virtualized data centers," *Cluster Computing*, vol. 12, no. 1, pp. 45–57, 2009.

[15] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven qos guarantees and optimization in clouds," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, ser. CLOUD '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 15–22.

[16] G. Jung, M. A. Hiltunen, K. R. Joshi, R. D. Schlichting, and C. Pu, "Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures," in *ICDCS*. IEEE Computer Society, 2010, pp. 62–73.

[17] S. Ferretti, V. Ghini, F. Panzieri, M. Pellegrini, and E. Turrini, "Qos-aware clouds," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. IEEE Computer Society, 2010, pp. 321–328.

[18] N. Huber, F. Brosig, and S. Kounev, "Model-based self-adaptive resource allocation in virtualized environments," in *SEAMS '11*. IEEE Computer Society, 2011.

[19] Y. O. Yazir, C. Matthews, R. Farahbod, S. Neville, A. Guitouni, S. Ganti, and Y. Coady, "Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, ser. CLOUD '10. IEEE Computer Society, 2010, pp. 91–98.

[20] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "Qos-aware replanning of composite web services," in *Proceedings of the IEEE International Conference on Web Services*, ser. ICWS '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 121–129. [Online]. Available: http://dx.doi.org/10.1109/ICWS.2005.96

[21] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[22] J. W. Eaton, *GNU Octave Manual*. Network Theory Limited, 2002.